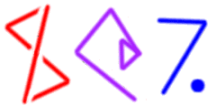
"How do I delete a line from a file?"

(Strategies for Lightweight Databases)

Mark Jason Dominus

Plover Systems Co.

`mjd-tpc-lwdb+@plover.com`



v1.2 (September, 2003)

FAQ

- `perlfaq5` says:

How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?

- This class will answer these questions

What We'll Do

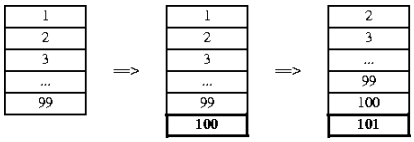
- Plain text files
  - The old 'copy the file' method
  - `seek` and indexing
  - `Tie::File`
- DBM files
  - `DB_File`
- Various applications and case studies

Text Files

- Text files are used all the time for lightweight databases
  - For example, Unix's `/etc/passwd` file
  - Apache's analogous password files
  - Databases and spreadsheets dumped into 'CSV' (comma-separated value) format
  - Server log files

Rotating Log File

- Here's a typical problem:
  - Append a line to the end of a log file
  - But the log file should contain only the most recent 100 lines
  - If it's longer than that, the old lines should be removed from the beginning



Deleting a User

- Another typical problem:
  - User `billg` has been fired
  - We want to remove his account
  - We should delete his entry from the password file

```
mjd:A2lJWJVp5BqDA
isi:A1gDcdPxmSOMY
tchrist:A3Jye3/wLzQNs
lenhard:A4z2KThzpHppE
gnat:A51FSA8JrmV6M
oznoid:A61i7deQ1D.82
rspier:B2lk7jM.0tjgk
billg:B35TsiJGzy/3w
layer:B6/E4Qdz9Dsss
maeda:KikFYFOSnGTwM
```


Copy the File

- The simplest and most often-cited solution is to copy the file
  - Make the changes as you write the copy
  - Then replace the original with the copy
- For example, deleting `billg`:

```
sub delete_user {
  my ($file, $target_user) = @_;
  open my $rfh, "<", $file or die ...;
  open my $wfh, ">", "$file.tmp" or die ...;
  while (<$rfh>) {
    my ($user) = split /:/;
    print $wfh unless $user eq $target_user;
  }
  close $rfh; close $wfh or die ...;
  rename "$file.tmp", $file or die ...;
}
```

- Or appending to a log file:

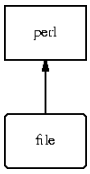
```
sub append_log {
  my ($file, @newrecs) = @_;
  open my $rfh, "<", $file or die ...;
  open my $wfh, ">", "$file.tmp" or die ...;
  my @recs = (<$rfh>, @newrecs);
  splice @recs, 0, @recs-$MAXRECS if @recs > $MAXRECS;
  print $wfh @recs;
  close $rfh; close $wfh or die ...;
  rename "$file.tmp", $file or die ...;
}
```


Copy the File

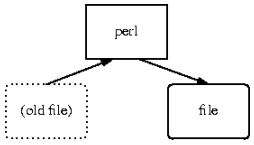
- Perl's `-i` option can make this easy:

```
perl -i -F: -lane 'print unless $F[0] eq "billg"' .users
```

- `-i` opens the original file for reading



- Then removes it
- The reopens the same name for writing



- Redirects standard output into the new file
- Data written to `STDOUT` is captured in the file

`-i.bak`

- Alternatively, you can have Perl leave behind a backup file

```
perl -i.bak -F: -lane 'print unless $F[0] eq "billg"' .users
```

- This is the same as `-i`
- Except Perl does not remove the original file
  - Instead, it renames it to `file.bak`
- If Perl crashes partway through, the old data is still available in `file.bak`
  - (Or if you decide you don't like the change.)

Using `-i` inside a program

```
perl -i.bak -F: -lane 'print unless $F[0] eq "billg"' .users
```

- That's all very well as a shell command
- What if you want to remove `billg` as part of a larger Perl program?
- Of course, one option is obviously:

```
system(qq{perl -i.bak -F: -lane
   'print unless \$F[0] eq "billg"' .users});
```


Using `-i` inside a program

- Using the `-i` facilities from inside a program requires a little trick
- The files that `-i` operates on are the ones named in `@ARGV`
- The special `$^I` variable holds the backup file suffix
  - (Empty string if no backup)
- To engage `-i`, set up `@ARGV` and `$^I` and run a `while <>` loop:

```
sub delete_user {
  my ($file, $target_user) = @_;
  local $^I = ".bak";
  local @ARGV = ($file);
  while (<>) {
    my ($user) = split /:/;
    print unless $user eq $target_user;
  }
}
```

- Now the opening and renaming are all implicit
- Use `local` so that `$^I` and `@ARGV` recover their old values when the function is done

Problems with `-i`

- For casual tasks, `-i` is very handy
- But if Perl crashes or the system goes down in the middle, the data is lost
- Even if Perl *doesn't* crash, the file is in an inconsistent state while it's being rewritten
- Hair-raising example:

```
perl -i.bak -F: -lane 'print unless $F[0] eq "billg"' /etc/passwd
```

- Suppose perl gets swapped out just after it renames `/etc/passwd`
  - Now the password file is empty
  - Anyone can log in with no password
- We need a more reliable strategy

Copy With Changes

- This is something like what `-i` does:

```
sub delete_user {
  my ($file, $target_user) = @_;
  open my $rfh, "<", $file or die ...;
  rename $file, "$file.bak" or die ...;
  open my $wfh, ">", $file or die ...;
  while (<$rfh>) {
    my ($user) = split /:/;
    print $wfh unless $user eq $target_user;
  }
  close $rfh; close $wfh;
}
```

- The problem is that the `rename` is too soon
  - We shouldn't replace the old contents with new so early
  - We should wait until the complete new file is in place

Copy With Changes

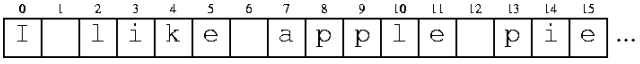
- This version (which we saw earlier) is safer:

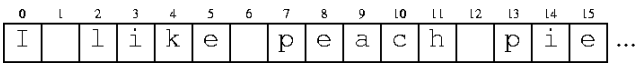
```
sub delete_user {
  my ($file, $target_user) = @_;
  open my $rfh, "<", $file or die ...;
  open my $wfh, ">", "$file.tmp" or die ...;
  while (<$rfh>) {
    my ($user) = split /:/;
    print $wfh unless $user eq $target_user;
  }
  close $rfh or die ...;
  close $wfh or die ...;
  rename "$file.tmp", $file or die ...;
}
```

- `rename` is guaranteed to be *atomic*:
  - At every instant, exactly one version of the file exists
  - If the function fails, or Perl crashes, the old file is untouched
  - At the moment the `rename` succeeds, the entire new file is in place
  - (Warning: `file.tmp` and `file` must be on the same filesystem)
- Why doesn't `-i` do it this way?
  - No good reason; coming in 5.10.

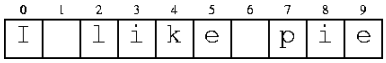
Essential Problem

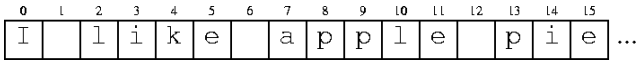
- Unix filesystems treat files like a sequence of bytes
- The basic operations are:
  - `read` a certain amount of data at the current position
  - `write` a certain amount of data at the current position
  - `seek` - adjust the current position
  - `truncate` the file to a certain length
- You can *overwrite* data in place:



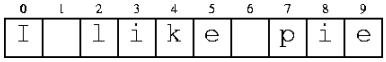


Essential Problem





- But there is no option to *insert* or *remove* data
  - To insert, you must copy the following data forward
  - To remove, you must copy the following data backward



- (Other OSes may support more powerful operations)

Essential Problem

- Moreover, byte-oriented operations are inconvenient for record-oriented programs
- Counting the number of bytes is easy:

```
my $n_bytes = -s $file;
```

- Counting the number of lines is hard:

```
open F, "<", $file;
while (<F>) {
  $n_lines++;
}
```

- Reading or writing at a certain byte position is easy:

```
seek F, $B_POSITION, SEEK_SET;
```

- Reading or writing at a certain line position is hard:

```
seek F, 0, SEEK_SET;  $REC = 1;
<F> until $REC++ >= $L_POSITION;
```

- The copy-the-file technique is simple, but it always pays the maximum possible cost

Fundamental Operations

Read

```
read(FH, my($buffer), $length);
```

- The standard I/O library enables reading by records
  - Data is read a block (4k or 8k) at a time into an internal buffer
  - `read` and `<...>` copy data out of the buffer

```
$record = <FH>;
```

Write

```
print FH $buffer;
```

- Note the opposite of `read` is *not* `write`; it's `print`

Truncate

```
truncate FH, $length;
truncate $filename, $length;
```


Seek

- `seek` adjusts the current position of a filehandle

```
use Fcntl ':seek';    # For SEEK_SET etc.
```

- Absolute position:

```
seek FH, $position, SEEK_SET;
```

- Relative position:

```
seek FH, $position, SEEK_CUR;
```

- Relative to the end of the file:

```
seek FH, $position, SEEK_END;
```

- `tell` returns the current absolute position:

```
my $position = tell FH;

# read, write, and seek FH here ...

seek FH, $position, SEEK_SET;
```

- This is guaranteed to put the handle back where it was at the time of the `tell`

Costs

- We'll see many different methods for searching and maintaining flat files
- They all have tradeoffs
- Some support quick searches
- Some support quick modifications
- There's always a tradeoff

	Copy the file
Add record	*S*
Delete record	*S*
Modify record	*S*

- Here *S* is the size of the file
- This means that it takes about twice as long to deal with a file that is twice as big.

Successful search	*S*/2
Unsuccessful search	*S*

- On average, we only have to search half the file if the record is there
- But the whole file if not

Adding Records

```
Add record              S
```

- With a plain flat text file, there's a shortcut for adding records
- Adding a record at the *end* of the file is very cheap

```
Append to beginning     S
Append to middle        S
Append to end           1
```

- The code looks like this:

```
sub add_user {
  my ($file, $new_user_data) = @_;
  local *F;
  open F, ">>", $file or return;
  print F $new_user_data, "\n";
}
```


Sorted Order

- If we keep the file in sorted order, searching is faster
- We can use a *binary search*
  - This is the method we use for searching the telephone book
- Idea:
  - Look at a record near the middle of the file
    - If the record is too early, look only at the last half of the file
    - If the record is too late, look only at the first half of the file
  - Repeat on successively smaller segments of the file
  - The standard `Search::Dict` module does this

Binary Search

- Binary search is notoriously difficult to code
  - There are a lot of funny edge cases
- If you write it yourself, test very carefully
- Or use `Search::Dict`
- Or the (carefully tested) code in your handouts

Binary Search

- This function gets a filehandle open to a sorted file
- It finds the first line in the file that is `ge $key`
- Returns that line and leaves `$fh` positioned at that line

```
sub search {
  my ($fh, $key) = @_;

  my ($lo, $hi) = (0, -s $fh);

  while (1) {
    my $mid = int(($lo + $hi)/2);

    if ($mid) {
      seek $fh, $mid-1, SEEK_SET;
      my $junk = <$fh>;
    } else {
      seek $fh, 0, SEEK_SET;
    }

    my $start = tell $fh;
    my $rec = <$fh>;
    return unless defined $rec;
    chomp $rec;

    if ($hi == $lo) {
      seek $fh, $start, SEEK_SET;
      return $rec
    }

    if ($rec lt $key) { $lo = $mid+1 }
    else              { $hi = $mid   }
  }
}
```

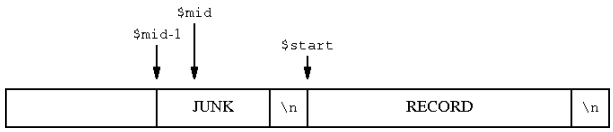
- This is `search1.pl` in your handout

Binary Search

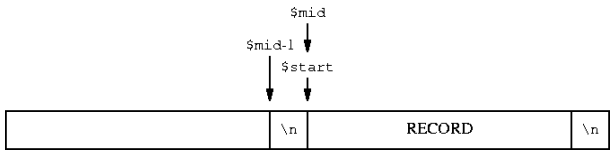
- What's with this?

```
if ($mid) {
  seek $fh, $mid-1, SEEK_SET;
  my $junk = <$fh>;
} else ...

my $start = tell $fh;
my $rec = <$fh>;
```

- Well, we want the record that starts at or after `$mid`
  - But `$mid` might point into the middle of a record



- We back up one space in case it *doesn't* point into the middle:



- Note: This trick only works when `length($/)` = 1

Binary Search

- `search3.pl` works for any value of `$/`
- It's similar to the innards of `Search::Dict`
- It uses binary search only to locate the *block* that contains the target
- Then it does linear search on the block
- It's about 75% slower than `search1.pl`
- Also, it might fail if any of the records are longer than a disk block
- The code is at the back of your book

Sorted Order

- Here's a benchmark result comparing linear search against binary:

```
          user  sys   total
Linear:   9.98  0.23  10.21
Binary:   0.01  0.04   0.05
```

- This is on ten randomly selected keys
- The target file contained 234,693 lines
- Here's 1000 searches with `search1.pl` and `search3.pl`:

```
   NULL:  0.00  0.00   0.00
Search1:  3.00  0.44   3.44
Search3:  5.58  0.39   5.97
```


Sorted Order

- `search2.pl` in your handout is like `search1.pl`, but a little more general
- It takes a search function that compares records
  - The function should return a negative value if the current record is too early
- `search2.pl` finds the first record in the file that is not 'too early'
- For example, if your file is the password file, sorted on field 2:

```
search(\*PASSWD,
       sub { my ($uid) = (split /:/)[1];
             $uid <=> 119 });
```

- This locates the first record whose UID is at least 119

Sorted Order

- The big drawback of sorted files is that they're hard to update
- You can't just append a new record at the end
- Comparison:

```
          Unsorted      Sorted

Lookup    Slow          Fast
Add       Fast          Slow
Delete    Slow          Slow
```

- An alternative is a hybrid approach
  - Have two files, one sorted, one unsorted
- For lookups, search the sorted file first, then the unsorted file
- To add records, append to the unsorted file
- Periodically merge the unsorted file into the sorted one

Modifying Records

- Modifying records in-place is tricky
- Because there might not be enough room for the new version
- Or the new version might not be big enough to fill all the space

Overwriting Records

- Suppose we are replacing a record with another of *exactly* the same length

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
b	i	l	l	g	:	B	3	5	T	s	i	J	G	z	y	/	3	w	\n	...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
B	I	L	L	G	:	B	3	5	T	s	i	J	G	z	y	/	3	w	\n	...

- Then we need not rewrite the entire file
- For example:

```
sub uppercase_username {
  my ($fh, $username) = @_;
  seek $fh, 0, SEEK_SET;
  while (<$fh>) {
    my ($u, $rest) = split /:/, $_, 2;
    next unless $u eq $username;
    seek $fh, -length($_), SEEK_CUR;
    print $fh uc($u);
    return;
  }
}
```

- We search the file as usual
- When we find the record we want, we back up and overwrite it in place

Bytes vs. Positions

- This looks innocuous, but it opens a 55-gallon drum of worms:

```
seek $fh, -length($_), SEEK_CUR;
```

- Here we wanted to back up to the beginning of the current record
- This won't always work
- Seek positions don't always correspond to character offsets
- Consider a DOS file:

```
I like pie\r\n
Especially apple.\r\n
```

- After reading the first record, `tell` is likely to return **12**
- But `$_` will contain `"I like pie\n"` (**11** characters)
  - The `\r\n` is translated to just `\n` on input
- The problem gets much worse with variable-length character encodings like UTF-8

Bytes vs. Positions

- I could probably talk all day about the various problems that come up
- So instead, we'll have one slide
- This *always* works, no matter what:

```
my $position = tell FH;
# read and write FH here ...
seek FH, $position, SEEK_SET;
```

- This *always* works, no matter what:

```
seek FH, 0, SEEK_SET;
```

- Bytes and characters are the same on Unix systems when files have 8-bit encodings
  - (Like ordinary text files, or files with ISO-8859 characters)
- Ditto for DOS/Windows systems **if** the filehandle is in *binary mode*:

```
binmode(FH);
```


Gappy Files

- If we need to modify variable-length records, we can do that
- Recall that the problems are:
  1. The new version of the record might not be big enough to fill all the space
  2. Or there might not be enough room for the new version in the old space
- (1) is easy to deal with: Just leave behind some padding characters
- (2) can't be dealt with; the record must move
  - Replace it with padding and put the new record at the end
- You also have to fix your search function to ignore the padding
- Example code is in `modify-in-place.pl`; example data in `MIP`

Gappy Files

- Searching first:

```
sub find {
  my ($fh, $key) = @_;
  seek $fh, 0, SEEK_SET;
  my $pos = 0;
  while (<$fh>) {
    chomp;
    next if /^\0*$/;
    if (index($_, $key) == 0) {
      seek $fh, $pos, SEEK_SET;
      return $_;
    }
    $pos = tell $fh;
  }
  return;
}
```

- This locates the first record that starts with `$key` and returns it
  - Also leaves `$fh` positioned at the start of that record
- The key here is to ignore any line that is all NUL characters
  - These represent 'gaps' from which data has been removed

Gappy Files

```
sub modify {
  my ($fh, $rec) = @_;
  my $pos = tell $fh;
  chomp(my $oldrec = <$fh>);
  seek $fh, $pos, SEEK_SET;

  if (length $oldrec == length $rec) {
    # easy case
    print $fh $rec;

  } elsif (length $rec < length $oldrec) {
    my $shortfall = length($oldrec) - length($rec);
    my $fill = "\0" x ($shortfall-1);
    print $fh $rec, "\n", $fill;

  ... continued ...
```

- In this case, the new record will fit in the old space
- Say we're changing `tchrist:A3Jye3/wLzQNs\n` to `tom:A3Jye3/wLzQNs\n`
  - We actually change it to `tom:A3Jye3/wLzQNs\n\0\0\0\n`
  - This is the same length
- `find` will ignore the `\0\0\0\n` 'gap'

Gappy Files

```
sub modify {

... continued ...

  } else { # New record is too big
    my $fill = "\0" x length($oldrec);
    print $fh $fill;
    seek $fh, 0, SEEK_END;  # New record goes at the end
    print $fh $rec, "\n";
  }
}
```

- Here the new record won't fit
  - We remove the old record entirely, and put the new one at the end
- Say we're changing `gnat:A51FSA8JrmV6M\n` to `torkington:A51FSA8JrmV6M\n`
  - We replace `gnat:A51FSA8JrmV6M\n` with `\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\n`
  - We put `torkington:A51FSA8JrmV6M\n` at the end of the file
- `find` will ignore the `\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\n` 'gap'
- A sufficiently clever implementation might put something else into the gap later

Gappy Files

- A variation: fill the gaps with newlines instead of NULs:

```
...
if (length $oldrec == length $rec) {
  # easy case
  print $fh $rec;
} elsif (length $rec < length $oldrec) {
  my $shortfall = length($oldrec) - length($rec);
  my $fill = "\n" x $shortfall;
  print $fh $rec, "\n", $fill;
} else { # New record is too big
  my $fill = "\n" x length($oldrec);
  print $fh $fill;
  seek $fh, 0, SEEK_END;  # New record goes at the end
  print $fh $rec, "\n";
}
...
```

- Then have `find` ignore blank lines
  - See `mip2.pl` in the handout

Fixed-Length Records

- If all the records are the same length, it becomes easy to modify them
  - The 'gappy files' code can dispense with the two hard cases
- Suppose the file has the following format:
  - 8-character username
  - 5-character user ID
  - 13-character encrypted password
  - 20-character surname
  - 18-character given name
- Each record is exactly 64 bytes

```
billg   666  xyPXkPQTjCjnUGates              William
mjd     119  A12FMTJGN5wpYDominus            Mark Jason
longuser123453OUbq7V0pV9doLongusername-EarwiggFeatherstonehaughs
larry   1    ..qQYY3gYY0yAWall               Larry
...
```

- Note that fields may now contain spaces or any other data

Fixed-Length Records

```
billg   666  xyPXkPQTjCjnUGates              William
mjd     119  A12FMTJGN5wpYDominus            Mark Jason
longuser123453OUbq7V0pV9doLongusername-EarwiggFeatherstonehaughs
larry   1    ..qQYY3gYY0yAWall               Larry
...
```

- Code to search for a user now looks like this:

```
sub find_user {
  my ($fh, $user) = @_;
  seek $fh, 0, SEEK_SET;
  while (read($fh, my($rec), 64) == 64) {
    my (@field) = unpack "A8 A5 A13 A20 A18", $rec;
    next unless $user eq $field[0];
    return @field;
  }
  return;
}
```

- `unpack` gets a *format string* that describes the data format
- `A8` is a space-padded alphabetic field of length 8
- Trailing spaces are trimmed off
- `a8` is similar, but NUL-padded (`"\0"`)

Fixed-Length Records

```
while (read($fh, my($rec), 64) == 64) {
  ...
}
```

- Alternative: Starting in 5.6.0, you may say

```
sub find_user {
  my ($fh, $user) = @_;
  seek $fh, 0, SEEK_SET;
  local $/ = \64;
  while ($rec = <$fh>) {
    my (@field) = unpack "A8 A5 A13 A20 A18", $rec;
    next unless $user eq $field[0];
    return @field;
  }
  return;
}
```


Fixed-Length Records

- Searching takes just as long as with variable-length records
- The win is when we want to modify a record

```
sub modify_user_data {
  my ($fh, $user, @newdata) = @_;
  unless (find_user($fh, $user)) {
    seek $fh, 0, SEEK_END;      # Move to end of file
  }
  print $fh pack "A8 A5 A13 A20 A18", $user, @newdata;
}
```

- Before, modifying was difficult in general
  - The only easy way was to copy the entire file
  - Now modifying in-place is cheap

Numeric Indices

- In some cases, we can speed up searching
- Suppose we want to look up usernames by UID
- Trick: Record number n holds information for user #n

```
root    0    PLM/VF.estSxwSuper user
larry   1    ..qQYY3gYY0yAWall                Larry
GARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEG
GARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEG
...
GARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEG
GARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEG
mjd     119  A12FMTJGN5wpYDominus              Mark
GARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEG
...
GARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEGARBAGEG
billg   666  xyPXkPQTjCjnUGates               William
...
```

- Locating a user is now very fast:

```
sub find_user_by_uid {
  my ($fh, $uid) = @_;
  seek $fh, $uid * 64, SEEK_SET;
  if (read($fh, my($rec), 64) == 64) {
    my (@field) = unpack "A8 A5 A13 A20 A18", $rec;
    return @field;
  }
  return;
}
```


`lastlog`

- Unix `lastlog` records are stored in this way

```
sub get_lastlog_info {
  my $user = shift;
  $user = getpwnam($user) if $user =~ /\D/;
  return unless defined $user;
  open L, "<", "/var/log/lastlog" or return;
  seek L, 292 * $user, SEEK_SET;
  return unless read(L, my($buf), 292) == 292;
  my ($time, $tty, $host) = unpack "i a32 a256", $buf;
  return ($time, $tty, $host);
}
```

- To use:

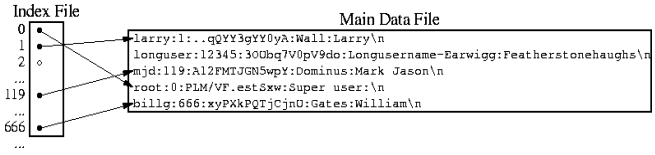
```
if (my ($time, $tty, $host) = get_lastlog_info(shift)) {
  print "$user on $tty from $host\n\tat ",
        scalar(localtime $time), "\n";
} else {
  print "$user never logged in\n";
}
```

```
% perl lastlog.pl hkang
hkang on pts/18 from evrtwa1-ar10-4-61-239-206.evrtwa1.dsl-ver
        at Sat Apr 19 15:45:56 2003

% perl lastlog.pl www
www never logged in
```

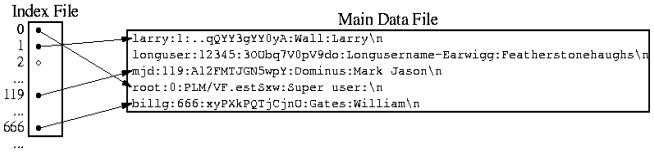
- Complete code in `lastlog.pl`

Indexing

- We can combine the speed of fixed-length records with the flexibility of variable-length records
- Idea: Make an *index* that records the position at which each record begins
- The index itself is fixed-length



Indexing



- To look up a user by UID:

```
sub find_user_by_uid {
  my ($fh, $index_fh, $uid) = @_;
  seek $index_fh, $uid * 10, SEEK_SET;
  if (read($index_fh, my($rec), 10) == 10) {
    my ($offset) = unpack "A10", $rec;
    seek $fh, $offset, SEEK_SET;
    my $record = <$fh>;
    return defined $record ? split /:/, $record : ();
  }
  return;
}
```


Indexing

- That's great, but where did the index file come from?
- Index files are easy to build:

```
sub build_index_for_users {
  my ($fh, $index_fh) = @_;
  seek $fh, 0, SEEK_SET;
  seek $index_fh, 0, SEEK_SET;
  truncate $index_fh, 0;       # Discard old index
  my $pos = tell $fh;
  while (<$fh>) {
    my $uid = (split /:/)[1];
    seek $index_fh, $uid * 10, SEEK_SET;
    print $index_fh, pack "A10", $pos;
    $pos = tell $fh;
  }
}
```


Void Fields

- There's a potential problem with the previous slide's code
- Suppose there is no user #2
- `find_user_by_uid` will go to the right place in the index
  - It will read 10 nonsense bytes
  - Then it will read from a nonsense position in the data file
- Solution: Fill nonsense fields with a special 'no such user' value:

```
sub build_index_for_users {
  my ($fh, $index_fh) = @_;
  seek $fh, 0, SEEK_SET;
  my $pos = tell $fh;
  my @position;
  while (<$fh>) {
    my $uid = (split /:/)[1];
    $position[$uid] = $pos;
    $pos = tell $fh;
  }
```

- (Continued...)

Void Fields

- (Continued...)

```
        seek $index_fh, 0, SEEK_SET;
        truncate $index_fh, 0;       # Discard old index
        for (@position) {
          if (defined) {
            print $index_fh pack("A10", $_);
          } else {
            print $index_fh "NoSuchUser";
          }
        }
      }
```

- `find_user_by_uid` then gets:

```
return if my $offset eq 'NoSuchUser';
```


Generic Text Indices

- Sometimes the index number is just the record number:

```
sub build_index_for_text {
  my ($fh, $index_fh) = @_;
  seek $fh, 0, SEEK_SET;
  seek $index_fh, 0, SEEK_SET;
  truncate $index_fh, 0;       # Discard old index
  my $pos = tell $fh;
  while (<$fh>) {
    print $index_fh, pack "A10", $pos;
    $pos = tell $fh;
  }
}
```


Packed Offsets

- Instead of making index numbers 10-byte strings, use 4-byte machine integers:

```
print $index_fh, pack "N", $pos;
```

- Instead of `"0 "`, we use `"\x00\x00\x00\x00"`
- Instead of `"1 "`, we use `"\x00\x00\x00\x01"`
- Instead of `"10 "`, we use `"\x00\x00\x00\x0a"`
- Instead of `"1000000000"`, we use `"\x3b\x9a\xca\x00"`
  - Benefit: Smaller index

`Tie::File`

- Perl 5.6.1 introduced a new module called `Tie::File`
- It makes a file look like an array
- Each line in the file becomes an array element
- Reading or modifying the array reads or modifies the file
- I wrote it because I didn't like the answer to the FAQ question:

How do I change one line in a file?

- The answer wasn't as helpful as I would have liked:

```
Those are operations of a text editor.
Perl is not a text editor.
```


`Tie::File` Examples

How do I change one line in a file?

```
tie @LINE, 'Tie::File', 'my_file.txt' or die ...;
for (@LINE) {
  if (/not a text editor/) {
    s/not/now/;
    last;
  }
}
```


`Tie::File` Examples

How do I delete a line in a file?

```
for $n (reverse 0 .. $#LINE) {
  if (is_snide_answer_to_FAQ($LINE[$n])) {
    splice @LINE, $n, 1;
  }
}
```

or:

```
my $spliced = 0;
for $n (0 .. $#LINE) {
  if (is_snide_answer_to_FAQ($LINE[$n - $spliced])) {
    splice @LINE, $n - $spliced, 1;
    $spliced++;
  }
}
```

or:

```
@snide = grep is_snide_answer_to_FAQ($LINE[$_])), 0..$#LINE;
for (reverse @snide) { splice @LINE, $_, 1 }
```


`Tie::File` Examples

How do I insert a line in the middle of a file?

```
for my $n (0 .. $#LINE) {
  if ($LINE[$n] =~ /<!--insert here-->/) {
    splice @LINE, $n+1, 0, $new_html_text;
    last;
  }
}
untie @LINE;
```


`Tie::File` Examples

How do I append to the beginning of a file?

```
tie @LOG, 'Tie::File', 'DrainC.log' or die ...;
unshift @LOG, $new_record1, $new_record2, @more_new_records;
```


`Tie::File` Examples

What Else?

```
tie @FILE, 'Tie::File', $file or die ...;

push @FILE, $new_last_record;
my $old_last_record = pop @FILE;
my $old_first_record = shift @FILE;

# Truncate or extend the file
$#FILE = 100;

# How long is the file?
$n_lines = @FILE;

# Overwrite the file
@FILE = qw(I like pie);
```


`delete_user` Revisited

- Here `$file` refers to a tied array instead of a filehandle:

```
sub delete_user {
  my ($file, $target_user) = @_;
  for my $n (0 .. $#$file) {
    my ($user) = split /:/, $file->[$n];
    next unless $user eq $target_user;
    splice @$file, $n, 1;
    last;
  }
}
```

- Or we might even use this:

```
sub delete_user {
  my ($file, $target_user) = @_;
  @$file = grep !/^$target_user:/, @$file;
}
```

- Wasn't that easy?
- Downside: The short version reads the entire file into memory

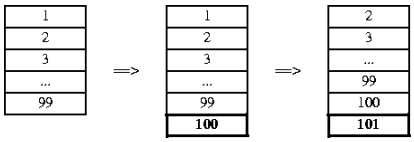
`uppercase_username` Revisited

- This is even easier:

```
sub uppercase_username {
  my ($file, $username) = @_;
  for (@$file) {
    last if s/^$username:/\U$username:/;
  }
}
```

- It's also efficient

Rotating Log File



- `Tie::File` solution:

```
tie @LOG, 'Tie::File', '/etc/logfile';

sub log {
  push @LOG, @_;
  my $overflow = @LOG - 100;
  splice @LOG, 0, $overflow if $overflow > 0;
}
```


Most Important Thing to Know About `Tie::File`

- It's for convenience, not performance
- I worked hard to make it reasonably fast
- But there's only so much that can be done

```
$FILE[0] =~ s/^x//;
```

- This is always going to have to read and rewrite the entire file
- `Tie::File` must perform reasonably well for many different types of applications
  - This means it's slower than code custom-written for a single application

General code is slower than special code

- For example:

```
sub uppercase_username {
  my ($file, $username) = @_;
  for (@$file) {
    last if s/^$username:/\U$username:/;
  }
}
```

- This builds and maintains an offset table in case you visit any of the early records again
- If you don't, the time spent is wasted
- The first version of `uppercase_username` didn't have to do that

Indexing with `Tie::File`

- You can use an in-memory hash as an index into a `Tie::File` file
- Here, `%index` maps usernames to record numbers:

```
my %index;
my $NEXT_UNREAD = 0;
sub find_user {
  my ($file, $user) = @_;
  my $rec;

  until (exists $index{$user}) {
    $rec = $file->[$NEXT_UNREAD];
    return unless defined $rec;
    my ($u) = unpack "A8", $rec;
    $index{$u} = $NEXT_UNREAD;
    $NEXT_UNREAD++;
  }
  return unpack "A8 A5 A13 A20 A18", $file->[$index{$user}];
}
```

- Each time this is called, it checks the index for the user you asked for
  - If the user is there, it uses `Tie::File` to retrieve the data quickly
  - If not, it scans the file until it finds what you wanted

Caching

- `Tie::File` also maintains an internal *read cache*
- If you try to read the same record twice, it comes from the read cache

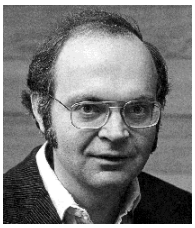
```
sub _fetch {
  my ($self, $n) = @_;

  # check the record cache
  { my $cached = $self->{cache}->lookup($n);
    return $cached if defined $cached;
  }
  ...
  $self->{cache}->insert($n, $rec)
    if defined $rec && not $self->{flushing};
  $rec;
}
```

- This is supposed to cut down on I/O
- You can limit the amount of memory used for the cache:

```
tie @FILE, 'Tie::File', $myfile, memory => 200000000;
```

- Default: 2 MiB

Caching



- Donald E. Knuth, a famous wizard, is fond of saying:

Premature optimization is the root of all evil.

- The cache is a good example of this
- Many common uses of `Tie::File` have a very low (or zero) cache hit rate

```
for (@FILE) { s/.../.../ }
unshift @FILE, items...;
```

- As a result, the cache just slows things down
- The next release of `Tie::File` will leave the cache disabled by default
  - It will enable the cache only if it believes this will help performance

`Tie::File` Modification

```
$FILE[$n] =~ s/this/that/;
```

- `Tie::File` knows how long each record is
- If you replace a record with one of the same length, it overwrites in place
- If the lengths differ, it must rewrite the tail of the file
  - It uses a block copy algorithm for this
- Truncating the file is easy:

```
$#FILE = 12;
```

- This locates the end of record 12 (if necessary) and truncates the file accordingly
- The general problem is very complicated and interesting
  - New improvements are always coming

Immediate vs. Deferred Writing

- By default, changes to the array are propagated to the file immediately
  - This is called *immediate writing*
- In some cases, this will be intolerably slow:

```
for (@FILE) {
  s/^/>> /;
}
```

- This modifies record 0, rewriting 0 .. 1000
  - Then it modifies record 1, rewriting 1 .. 1000
  - Then it modifies record 2, rewriting 2 .. 1000
  - This is intolerably slow

Deferred Writing

- If performance is more important than immediate writing, you may disable it:

```
my $f_obj = tied(@FILE);

$f_obj->defer;

for (@FILE) {
  s/^/>> /;
}

$f_obj->flush;
```

- All writing is done in memory until you call `->flush`
  - (Or until the memory limit you specified is exceeded.)
  - Then all the writing is done in one batch

```
tie @FILE, 'Tie::File', $myfile, dw_size => 500000;
```

- Default: Whatever the `memory` limit is

Autodeferring

```
for (@FILE) {
  s/^/>> /;
}
```

- Loops like this are common
- `Tie::File` detects these and enables deferred writing *automatically*
- Then disables it again when you're done
- Unless you don't want that:

```
tie @FILE, 'Tie::File', $myfile, autodefer => 0;
```


Miscellaneous Features

- Read-only mode:

```
use Fcntl 'O_RDONLY';
tie @FILE, 'Tie::File', $myfile, mode => O_RDONLY;
```

- Change the record separator string:

```
tie @FILE, 'Tie::File', $myfile, recsep => ";;";
```

- Tie an open filehandle:

```
tie @FILE, 'Tie::File', \*STDIN, mode => O_RDONLY;
```

- Lock the file:

```
use Fcntl ':flock';
(tied @FILE)->flock(LOCK_EX);
```

- (Locking is another advantage over `DB_File`)

DBM

- Perl's `tie` feature is a generalization of *DBM files*
  - (*DBM* is short for *Data Base Manager*, I think)
- Basic idea: A Perl data structure is backed by a disk file
  - Reading the data structures reads the file
  - Modifying the data structures writes the file
- This first appeared in perl 3

```
dbmopen %hash, $filename, $permissions;
dbmclose %hash;
```


DBM

```
dbmopen %hash, $filename, $permissions;
```

- There are several different libraries than can handle this association
- Which one did `dbmopen` use?
  - Whichever one was compiled into Perl
  - If you want to copy data from an `NDBM` file into an `ODBM` file, too bad
- This was one of the major motivations for the Perl 5 module system
- New syntax:

```
tie %hash, 'Package', ARGS...;
```

- The `Package` is a module responsible for implementing the association
- `dbmopen %hash, $file` is now emulated as

```
tie %hash, 'AnyDBM_File';
```

- `AnyDBM_File` tries several popular modules until it finds one that works

Common DBM Implementations

- There are five widely-used DBM libraries
  - `ODBM_File` uses the original DBM library, called `libdbm` (1979)
  - `NDBM_File` uses an improved version called `libndbm` (1985)
  - `GDBM_File` uses the GNU project library, `libgdbm` (1990?)
  - `SDBM_File` uses a new version called `libsdbm` (1991)
  - `DB_File` uses the Berkeley DB library `libdb` (1993)
- When you build Perl, it looks for each of these
  - It constructs and installs the `tie` modules for the ones you have
  - Exception: Perl comes with `libsdbm`, so you *always* have `SDBM_File`

What DBM Does

- DBM libraries store data in a hashed database
- It's like a Perl hash, but on the disk
- Advantage over plain text files:
  - Lookup is very fast
  - (Insertion is much less fast because data might have to be moved around)
- Disadvantage:
  - The file is full of binary gibberish

What DBM Does

- DBM libraries provide functions for storing, fetching, and generating lists of keys
- The `tie` modules provide a glue layer between these libraries and Perl

```
MODULE = SDBM_File      PACKAGE = SDBM_File     PREFIX = sdbm_
#define sdbm_FETCH(db,key) sdbm_fetch(db->dbp,key)
```

- These libraries are an attempt to keep a hash on the disk
  - Just as `Tie::File` is an attempt to keep an array on the disk
  - As with `Tie::File`, there are a lot of interesting tradeoffs to be made

Small DBMs: `ODBM`, `NDBM`, and `SDBM`

- These all have the same major drawback:
- The amount of data is limited
- Typically, key size + value size must be less than about 1KB for each key

```
use Fcntl 'O_RDWR', 'O_CREAT';
use SDBM_File;

tie %h, "SDBM_File", "/tmp/sdbm", O_RDWR|O_CREAT, 0666
    or die $!;
$h{ouch} = "-" x 1024;
print "ok\n";
```

- Nope:

```
sdbm store returned -1, errno 22, key "ouch" at sdbm_fail line
```

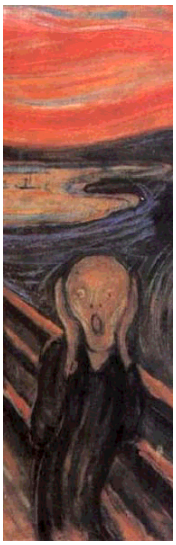
Small DBMs: ODBM, NDBM, and SDBM

- Another problem is that these databases use *sparse storage*
- The hash isn't represented very efficiently on the disk

```
# Keys   File extent  Space used
          (ls -l)     (ls -s)

     1         1024        8
     2         2048        8
     4         4096        8
     8         8192       48
    16       120832      120
    32       245760      208
    64       441344      296
   128      4251648      456
   256     12701696     1456
   512     21091328     2320
  1024     33284096     4128
  2048    536668160    11592
  4096   1065409536    22272
```

- Even though we're not storing that much data, the file extents get huge
- Many systems can't handle a file with an extent greater than 2GiB.
- On these systems, `O/N/SDBM` are severely limited in the amount of data they can store
- (This output produced by `sdbm_test.pl` in your handout)

GDBM



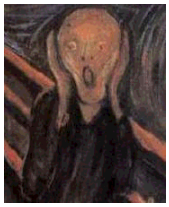
- `GDBM` does not have these data size problems
- I used to use it all the time
- Now I don't; here's why
- In 1998 I was using it for a web user database for a major client
  - The key was the user name and the value was the user's information
- We had about 320,000 registered users
- One day, the `firstkey` and `nextkey` routines stopped producing all the keys
  - They would generate about 1,700 of the usernames and then stop
  - I couldn't get the list of our users!

GDBM

- I sent a detailed bug report to the GNU folks, offering to do whatever I could to help
- The reply said:

```
I have heard of this happening before.  I was not able to
find out why.  Do you have a backup of earlier versions so
you can get most of your keys out?  If so, you might try to
recover by moving to DB-2.? routines.  They are still being
updated an developed.  gdbm has not had any active
development in years.
```

- So I restored what I could from the backup tapes
- I switched to Berkeley DB
- I have not used `GDBM` since



DB_File

- The best choice
- Berkeley DB was good ten years ago and it has gotten better
- Basic usage is very simple:

```
use DB_File;
tie %hash, 'DB_File', $file or die ...;
```

- Optional arguments:

```
tie %hash, 'DB_File', $file, O_RDONLY;

tie %hash, 'DB_File', $file, O_CREAT | O_RDWR;

tie %hash, 'DB_File', $file, O_CREAT | O_RDWR, 0666;
```

- Now use `%hash` just like any other hash

Indexing Revisited

- Here's a hybrid approach to indexing
- The bulk of the data will be in a plain text file
- But the plain text file takes too long to search
- So we'll also have a DBM file that records record byte offsets
- Then we can locate records quickly
- Complete example code is in `indexed.pl`

Indexing Revisited

- Searching will be fast:

```
sub find {
  my ($fh, $dbm, $key) = @_;
  my $offset = $dbm->{$key};
  seek $fh, $offset, SEEK_SET;
  my $rec = <$fh>;
  return $rec;
}
```

- `$fh` is a filehandle on the (plain text) data file
- `$dbm` is a reference to the DBM hash with the offset information
- `$key` is the key we want to look up
- We get the offset information from the DBM hash
- Seek the filehandle to the right position in the text file
- Read the right record instantly

Indexing Revisited

- Where did the offset information come from?

```
sub make_index {
  my ($fh, $dbm, $key_function) = @_;
  seek $fh, 0, SEEK_SET;
  %$dbm = ();
  my $pos = 0;
  while (<$fh>) {
    chomp;
    my $key = $key_function->($_);
    $dbm->{$key} = $pos;
    $pos = tell $fh;
  }
}
```

- `$fh` and `$dbm` are as before
- `$key_function` takes a record from the file and says what the key should be
- If the DBM eats the index, your homework is still intact

Indexing Revisited

- A typical use:

```
use DB_File;
tie %by_name, 'DB_File', "./pw_aux", O_CREAT|O_RDWR, 0666
  or die $!;
open PASSWD, "<", "/etc/passwd" or die $!;

make_index(\*PASSWD, \%by_name,
           sub { (split /:/, $_[0], 2)[0] },
          );
```

- The key function here takes a password file line and extracts the username
- We only need to call `make_index` once
  - (Until the password file changes)
- After that, we can get as many fast lookups as we want:

```
print find(\*PASSWD, \%by_name, "mjd");
```

```
mjd:x:119:100:Mark Jason Dominus:/home/mjd:/bin/bash
```


Indexing Revisited

- We can also build multiple indices:

```
open PASSWD, "<", "/etc/passwd" or die $!;
tie %by_name, 'DB_File', "./pw_aux", O_CREAT|O_RDWR, 0666
  or die $!;
tie %by_uid, 'DB_File', "./pw_uid", O_CREAT|O_RDWR, 0666
  or die $!;

make_index(\*PASSWD, \%by_name,
           sub { (split /:/, $_[0], 2)[0] },
          );

make_index(\*PASSWD, \%by_uid,
           sub { (split /:/, $_[0], 3)[1] },
          );

print find(\*PASSWD, \%by_name, 'mjd');
print find(\*PASSWD, \%by_uid, 119);
```

- (All this also works with untied hashes)

Ordered Hashes

- `DB_File` actually supports three different file types

```
tie %hash,  'DB_File', $file, O_CREAT|O_RDWR, 0666, $DB_HASH;
tie %hash,  'DB_File', $file, O_CREAT|O_RDWR, 0666, $DB_BTREE;
tie @array, 'DB_File', $file, O_CREAT|O_RDWR, 0666, $DB_RECNO;
```

- The default is `DB_HASH` which we've seen already
- `DB_RECNO` associates a plain text file with an array
  - But `Tie::File` may be preferable, for a number of reasons
  - (See the handout)
- `DB_BTREE` uses a different data structure called a *B-tree*
  - Also called a *VSAM file* by big-iron types
  - Unlike a hash, it keeps the records *in order*
  - For very large databases (1,000,000 records) lookup may be slower than hashes

Ordered Hashes

```
tie %hash, 'DB_File', $file, O_CREAT|O_RDWR, 0666, $DB_BTREE;
```

- By default, the ordering is lexicographic:

```
for (qw(red orange yellow green blue violet)) {
   $hash{$_} = length;
}

print join(" ", keys %hash), "\n";
```

```
blue green orange red violet yellow
```

- You may specify an alternative ordering:

```
use DB_File;

my $rev_btree = DB_File::BTREEINFO->new();
$rev_btree->{compare} =
  sub {
    my ($a, $b) = @_;
    reverse($a) cmp reverse($b)
  };

tie %hash, 'DB_File', $file, O_CREAT|O_RDWR, 0666, $rev_btree;
```

```
red orange blue green violet yellow
```


Partial Matching

```
blue    4
green   5
orange  6
red     3
violet  6
yellow  5
```

- Because the keys in a B-tree are in order, you can do limited partial matching
- As with `Search::Dict`, you can look for the first key that begins with some string

```
my $db = tied %hash;
my $k = "g";
$db->seq($k, $v, R_CURSOR);
print "$k => $v\n";
```

```
green 5
```

- Actually it produces the first key that is greater than or equal to `$k`
  - Under the appropriate comparison
- If there is no such key, `$k` is unchanged, `$v` is `undef`, and `seq` returns **true**

Sequential Access

- `->seq` provides generic sequential access to the keys
  - In the user-defined order
- Note: C-style `for` loops coming up
- To scan the keys forwards:

```
my $db = tied %hash;
my ($k, $v, $fail);
for ($fail = $db->seq($k, $v, R_FIRST) ;
         ! $fail ;
         $fail = $db->seq($k, $v, R_NEXT)
    ) {
  print "$k => $v\n";
}
```

- Or backwards:

```
for ($fail = $db->seq($k, $v, R_LAST);
         ! $fail;
         $fail = $db->seq($k, $v, R_PREV)
    ) {
  print "$k => $v\n";
}
```

- Or just the keys between $a and $b:

```
$k = $a;
for ($fail = $db->seq($k, $v, R_CURSOR);
         ! $fail && $k le $b;
         $fail = $db->seq($k, $v, R_NEXT)
    ) {
  print "$k => $v\n";
}
```


Filters

- Suppose you want to store complex data structures in a `DB_File`
- This doesn't work:

```
$hash{numbers} = [1, 4, 2, 8, 5, 7];
```

- The array is converted to a string, and the *string* is stored:

```
print $hash{numbers}, "\n";
ARRAY(0x8118d9c)

$aref = $hash{numbers};
print "@$aref\n";

Can't use string ("ARRAY(0x8118d9c)") as an ARRAY ref
    while "strict refs" in use...
```

- This is a drawback of all DBM implementations
  - And indeed of the Unix operating system
  - There's no OS support for storing anything except a lifeless byte sequence

Filters

- We could use a serialization module like `Storable`
  - It will convert arbitrary values to strings, and back:

```
use Storable;
$hash{numbers} = freeze [1, 4, 2, 8, 5, 7];

$aref = thaw $hash{numbers};
print "@$aref\n";

1 4 2 8 5 7
```

- This is kind of a pain
- `DB_File` will do it automatically:

```
my $db = tied %hash;
$db->filter_store_value(sub { $_ = Storable::freeze($_) });
$db->filter_fetch_value(sub { $_ = Storable::thaw($_) });
```

- Now this works:

```
$hash{numbers} = [1, 4, 2, 8, 5, 7];
$aref = $hash{numbers};
print "@$aref\n";
```

- `freeze` and `thaw` are called automatically
  - Note that the filters use `$_` for input and output
- Similarly, `filter_store_key` and `filter_fetch_key`

`BerkeleyDB`

- The Berkeley DB library has many other fascinating features
  - Not all are available through `DB_File`
- For example, there is a `DB_Queue` file type
  - This is like an array
  - But it is optimized for `push` and `shift` operations
  - You might use this to store a log file
  - When the log file exceeds a certain size, you shift the old records off the front
- There is an option to keep the values for duplicate keys in a user-defined order
- It supports transactions
- The `BerkeleyDB` module provides interfaces to this functionality
- It's worth skimming through the manual
- Check out `http://www.sleepycat.com/docs/reftoc.html` for a tutorial and overview

Thank You!

- Any questions?

Bonus Slides

- Classes change from year to year
- Some things move in, others come out
- There's never enough time to cover all the material I'd like to
- But you may as well see the deleted slides

`Tie::File` Internals

- Inside, `Tie::File` uses a combination of several of the techniques we've seen
- It maintains an offset table internally

```
$z = $FILE[57];
```

- This checks the offsets table for `$offsets[57]`
  - If it's already present, `Tie::File` seeks to the right location and reads the record
  - If not, `Tie::File` scans from the last known position up to line 57

```
sub _fetch {
  my ($self, $n) = @_;
  ...

  if ($#{$self->{offsets}} < $n) {
    return if $self->{eof};
    my $o = $self->_fill_offsets_to($n);
    # If it's still undefined, there is no such record,
    # so return 'undef'
    return unless defined $o;
  }

  my $fh = $self->{FH};
  # we can do this now that offsets is populated
  $self->_seek($n);
  my $rec = $self->_read_record;
  ...

  $rec;
}
```


Multiple Values

- Unlike a hash, a B-tree may store more than one value per key
- To enable this, use `R_DUP`:

```
my $dup_btree = DB_File::BTREEINFO->new();
$dup_btree->{flags} = R_DUP;
tie %hash, 'DB_File', $file, O_CREAT|O_RDWR, 0666, $dup_btree;

while (<FRUITS>) {
   my ($color, $fruit) = split / /, $_, 2;
   $hash{$color} = $fruit;
}
```

- Ordinary hash assignment actually stores the new value *in addition to* the old one
- Hash retrieval recovers only the **first** stored value
- But `->seq` will recover *all* the values:

```
$k = "red";
for ($fail = $db->seq($k, $v, R_CURSOR);
     ! $fail && $k eq "red";
     $fail = $db->seq($k, $v, R_NEXT)) {
  print "$k: $v\n";
}

red: apple
red: cherry
red: strawberry
red: raspberry
```

Multiple Values

- For a user-defined comparison, 'identical' keys might not be exactly the same
- Suppose the comparison is case-insensitive:

```
my $my_btree = DB_File::BTREEINFO->new();
$my_btree->{flags} = R_DUP;
$my_btree->{compare} = sub { lc $_[0] cmp lc $_[1] };
tie %hash, 'DB_File', $file, O_CREAT|O_RDWR, 0666, $my_btree;
```

- Then `Red`, `red`, and `RED` are all considered 'the same'

```
for (['Red', 'apple'], ['red', 'cherry'],
     ['RED', 'strawberry'], ['blUe', 'grape']) {
   my ($key, $value) = @$_;
   $hash{$key} = $value;
}
```

- Only the first of these three is actually stored
- The hash interface will report the duplicate keys:

```
print join(" ", keys %hash), "\n";
```

```
blUe Red Red Red
```


Multiple Values

```
$my_btree->{compare} = sub { lc $_[0] cmp lc $_[1] };

%hash = (Red => 'apple', red => 'cherry', RED => 'strawberry',
         blUe => 'grape');
```

- Since the keys are insensitive, the hash interface can't distinguish them:

```
print "$hash{blUe} $hash{red} $hash{Red} $hash{RED}\n";
```

```
grape apple apple apple
```

- But `->seq` can recover the values:

```
my $db = tied %hash;
$k = $START = 'red';
for ($fail = $db->seq($k, $v, R_CURSOR);
     ! $fail;
     $fail = $db->seq($k, $v, R_NEXT)) {
  print "$k: $v\n";
}
```

```
Red: apple
Red: cherry
Red: strawberry
```


Multiple Values

- There are some special methods for dealing with duplicate keys
- This recovers a list of the values associated with the given key:

```
@a = $db->get_dup($Key);
```

- This is a count of the number of appearances of the key:

```
$n = $db->get_dup($Key);
```

- You can checks to see if the key is associated with a certain value
  - If the pair is found, this returns **false** and positions the cursor at the specified pair

```
$failed = $db->find_dup($Key, $Value);
```

- You can then iterate over preceding or following key-value pairs with `->seq`
- You can delete just one key-value pair:

```
$failed = $db->del_dup($Key, $Value);
```

- This returns **false** on success